ECM2433 TRAFFIC SIMULATION PROGRAM REPORT

Instructions

Import the files and then execute compileSim.sh file using 'bash compileSim.sh'.

Program Design

Following the specification, I have created a C program that simulates cars passing through an intersection. The user can enter the parameter values and then execute the shell script. The main function is located in 'runSimulations'. The program takes the parameter values entered through the 'argv' array. The parameters are then converted to from strings to integers and stored in variables.

The runSimulations function contains a main function which accepts parameters from the command line. The expected parameters are the arrival rates of the vehicles and the length of time that the signals are green. In total, there are 5 parameters including the 'argc' integer. Therefore, if the user enters any other number of parameters an error will occur. If the signal times are not greater than 1 and arrival rates are not greater than zero, an error will occur too.

The next step is creating the random number generator, by making use of the gsl library.
I have used the GNU Scientific Library (GSL) for producing random numbers. The library provides a better standard of randomness than the standard C library. I have seeded the generator with time to ensure a higher quality of randomness. I have implemented the 'gsl_rng_uniform_pos' function which returns a positive double in the range of (0,1).
I have followed the official documentation steps. A design choice here was to use a local generator instead of a global generator. I did this as we were advised not to use global variables. However, this required me to pass the generator as a parameter to those functions that required it.

I have used an array to store the results of a simulation. The array is initialised with values of zero. The array is passed to runOneSimulation, where the results that need to be stored in the array are calculated. Since arrays in C are passed by reference, the updated array can be easily accessed in the main function. Therefore, there is no need to return the updated array. I used an array due to this reason. Additionally, the number of elements in the array do not change so
For each single simulation the results are added to the respective element in the array. To get the average of the results, the results are divided by 100.

The main function calls the runOneSimulation function 100 times. I used a for loop to call the function repeatedly as the number of iterations are known.
The runOneSimulation function initialises a number of variables that are required to calculate results and perform the simulation.
I have used a linked list to simulate a queue of cars on the road. I chose to use a linked list over an array as it would be inefficient to use an array for a list that can change in size. An array would require additional resources to accommodate a change in size. One would need to create a new array of the desired size and copy the elements from the old array into the new array. Doing this repeatedly would be very expensive. Furthermore, arrays require contiguous memory so the memory allocation would not be efficient. If there are many cars

joining the queue then storing such a large list in contiguous memory would not be desired. A linked list is ideal as it does not require contiguous memory. It can also accommodate changes in the size of the list very easily.
I have defined the structure of the linked list using a 'struct'. The car at the head and tail of the queue will be defined for each queue. A standard queue structure is suitable for this simulation as the cars will follow a first in first out pattern.

I have used a while loop to perform the actions in each timestep. A for loop was not used as the number of total timesteps is not known at the start. The conditions for the while loop test whether the queues are empty or whether the 500 iterations are completed.

Within the loop, there are broadly two categories of actions. One is related to handling the signals, and the second is for managing cars.

There are two states of a traffic signal – red and green. To represent these states, I have used integers 0 and 1. 0 represents red, and 1 represents green. I chose to use integers instead of defining booleans as there would be less memory used. I also chose not to use characters, in case I need to do integer comparisons between the two states or signals.

If a signal is green, then a car will exit the queue in that timestep. The car is dequeued and the queue is modified. There is a check to see whether any cars are left in the queue.
I also check whether it is time for the signal to turn red or green by checking the time remaining. The time remaining is decremented and then checked to see if it is equal to zero. The states of the signals are then made the opposite if the condition is met.

Whether cars are added to the queue is decided by a random number and the rate of arrival. Using GSL, a random double is returned. The double is then compared to the inputter rate of arrival. If the random number is less than the rate of arrival, then a car will arrive into the queue. The user enters a percentage arrival rate between 0 and 100. However, the random number library returns a uniformly distributed double between 0 and 1. Therefore, I divide the arrival rate by 100 to get a number in the range of (0,1), so that I can compare the numbers.

After 500 iterations are complete, the queue still needs to clear. An if condition is used to check whether the queue is empty or not. If empty, then the time it took for the queue to clear is calculated.

I have also used header files to make the program structure and format better and cleaner. The header files have '#includes', function prototypes and structure definitions. I have used two structs, one to represent cars and the second to represent queues. The cars are the nodes in the queue.

I have created a separate file named 'queue.c' to deal with the management of the queue. The functions deal with creating a new queue, adding and removing a node from the queue, and creating a new node.

I have used the malloc function to safely allocate memory and create a pointer to that memory space. I have also made use of the free function, to safely deallocate memory allocated by malloc.

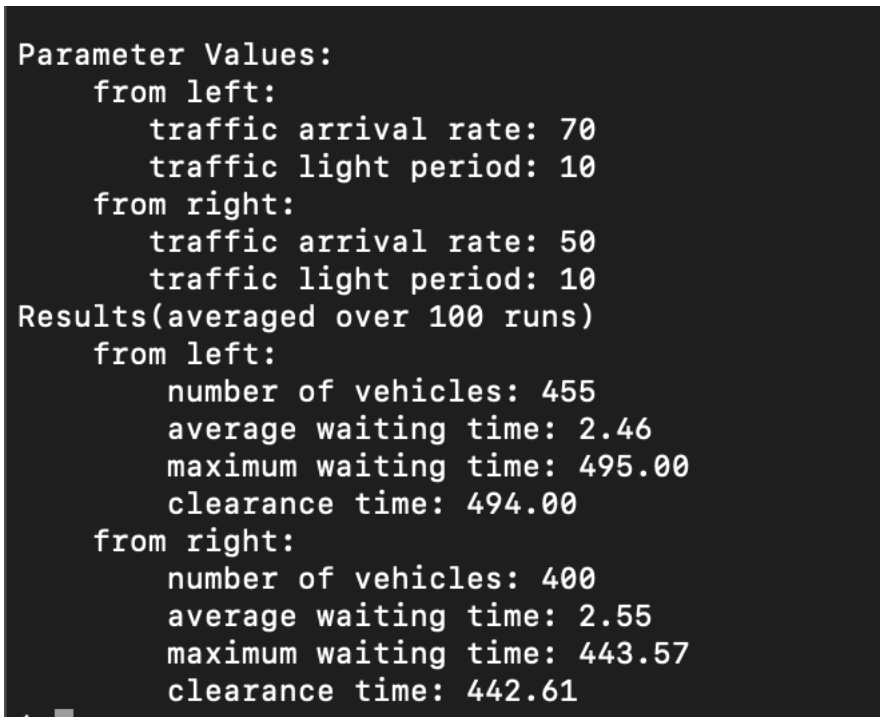
I have also used the extern keyword so that functions can be accessed by other files can utilise their functionality.

Initially, I created a structure for signals too. This would make it easier to expand the program and more signals later to create a more complex simulation.
I also defined more struct members for each struct. For example, I created members for average and maximum wait times inside the queue struct. However, I then decided to remove those members and defined the variables directly in runOneSimulation. I did this as it would make the program faster and more efficient. If I need to increase the functionality of the program later by creating more queues and signals, I will be able to add the necessary code.

Detected Errors

The average waiting time occasionally displays not a number. I tried changing the casting of variables, but I was unable to locate the cause of the error.

Experiment



These are the results from a simulation. The number of vehicles are higher in the left queue as it has a higher arrival rate. The average wait time is similar as both queues have the same traffic light period. The max waiting time is higher in the left queue as there more cars in the left queue due to the higher arrival rate. The clearance time is higher in the left queue as there are more cards in the left queue.